Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming

Lucas Davi, Christopher Liebchen,
Ahmad-Reza Sadeghi
CASED/Technische Universität Darmstadt, Germany
Email: {lucas.davi,christopher.liebchen,
ahmad.sadeghi}@trust.cased.de

Kevin Z. Snow, Fabian Monrose
Department of Computer Science
University of North Carolina at Chapel Hill, USA
Email: {kzsnow,fabian}@cs.unc.edu

***Abstract*—Until recently, it was widely believed that code randomization (such as fine-grained ASLR) can effectively mitigate code reuse attacks. However, a recent attack strategy, dubbed just-in-time return oriented programming (JIT-ROP), circumvents code randomization by disclosing the (randomized) content of many memory pages at runtime. In order to remedy this situation, new and improved code randomization defenses have been proposed.**

The contribution of this paper is twofold: first, we conduct a security analysis of a recently proposed fine-grained ASLR scheme that aims at mitigating JIT-ROP based on hiding direct code references in branch instructions. In particular, we demonstrate its weaknesses by constructing a novel JIT-ROP attack that is solely based on exploiting code references residing on the stack and heap. Our attack stresses that designing code randomization schemes resilient to memory disclosure is highly challenging. Second, we present a new and hybrid defense approach, dubbed Isomeron, that combines code randomization with execution-path randomization to mitigate conventional ROP and JIT-ROP attacks. Our reference implementation of Isomeron neither requires source code nor a static analysis phase. We evaluated its efficiency based on SPEC benchmarks and discuss its effectiveness against various kinds of code reuse attacks.

I. Introduction

Code reuse attacks, such as return-oriented programming (ROP) [44, 48], are predominant attack techniques extensively used to exploit vulnerabilities in modern software programs. ROP attacks hijack the control-flow of an application by maliciously combining short instruction sequences (gadgets) residing in shared libraries and the applications executable, and circumvent protection mechanisms such as data execution prevention (DEP or W$\oplus$X). Today, ROP remains a widely used attack strategy for exploiting vulnerabilities of software programs on commodity PC platforms (e.g., Internet Explorer [24], Adobe Reader [9]) as well as mobile devices based on ARM processors (e.g., Safari Browser Jailbreak [17]).

One class of mitigation techniques against return-oriented programming is *address space layout randomization (ASLR)*, currently a standard defense technique enabled on commodity operating systems [22, 36, 52]. ASLR randomizes the base addresses of code and data segments thereby randomizing the start addresses of each ROP sequence that the adversary attempts to invoke. However, due to the low randomization entropy on 32 bit systems, brute-force attacks can reverse conventional ASLR [45]. More importantly, a memory disclosure vulnerability that reveals runtime addresses (e.g., a function pointer) can be exploited to bypass ASLR since only the base address of a segment is randomized. Today, memory disclosure vulnerabilities are frequently exploited in state-of-the-art exploits [42].

To overcome the deficiencies of conventional ASLR, a number of *fine-grained* ASLR schemes have emerged that apply randomization to the code structure at different granularity, e.g., function-level or instruction location [14, 21, 26, 39, 55]. However, as shown recently, a new attack strategy, just-in-time return-oriented programming (JIT-ROP), can be used to undermine fine-grained ASLR [47]. It exploits the implicit assumption of fine-grained ASLR schemes that the adversary has to perform (offline) static analysis on the target application to identify useful ROP gadgets. JIT-ROP attacks use a *single* leaked runtime address to disassemble the content of hundreds of memory pages and generate ROP exploits on-the-fly. In fact, JIT-ROP attacks prominently show the importance of memory disclosure, posing design challenges on code randomization.

Goal and contributions. Our goal is to tackle the problem of constructing a runtime software diversifier resilient to traditional ROP and JIT-ROP attacks. Our main contributions are as follows:

Bypassing a state-of-the-art randomization scheme: Based on our analysis of ROP and JIT-ROP attacks, we evaluate the effectiveness of a recently proposed fine-grained ASLR scheme, dubbed Oxymoron, claimed to be secure against JIT-ROP attacks [3]. We developed a novel JIT-ROP attack that efficiently bypasses Oxymoron. We show the feasibility of our attack by crafting a real-world exploit that incorporates the restrictions of the proposed mitigation, but, nevertheless, gains arbitrary code execution.

Novel runtime diversifier: We present a novel defense, called Isomeron, that makes fine-grained randomization resilient to conventional ROP and JIT-ROP attacks. Our mitigation is

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23262

based on the idea of combining execution-path randomization with code randomization which – as we will show – exponentially reduces the success probability of the adversary to predict the correct runtime address of a target ROP gadget.

Proof-of-concept and evaluation: We instantiate our solution Isomeron using a new dynamic binary instrumentation framework which we specifically developed to realize our pairing of code and execution-path randomization. Our instrumentation framework (i) instruments all call, return, and jump instructions that a program execute during its lifetime, (ii) provides the ability to modify existing and insert new instructions into the instruction stream of the application at any time, and (iii) does not require access to the source code or debugging symbols of an application. We evaluated our prototype of Isomeron based on SPEC benchmarks, and describe its effectiveness against different kinds of code reuse attacks.

II. Bypassing Code Randomization with JIT-ROP

In this section, we briefly recall traditional ROP and JIT-ROP attacks, as well as (fine-grained) ASLR solutions.

A. Basics

The goal of a return-oriented programming (ROP) attacks is to hijack the intended execution flow of an application and perform malicious operations without injecting any new code. To subvert the execution flow, an adversary needs to identify a vulnerability in the target application. A typical example is a buffer overflow error on the stack [2] or heap [12]. Such vulnerabilities allow the adversary to write data beyond the memory space reserved for a buffer or a variable, so that critical control-flow information (e.g., a function pointer) can be overwritten.

In many real-world exploits the attack payload is embedded in a file that is processed by the target application, e.g., a HTML file to exploit a browser bug, or a PDF file to attack a PDF viewer. In a ROP attack, the payload consists of control data (pointers), where each pointer refers to a code sequence in the address space of the target application. The adversary then combines these code sequences to form gadgets where each gadget is responsible for performing a well-defined task, such as addition or loading from memory. Typically, the adversary deploys static analysis tools to identify useful code sequences before launching the ROP attack.

Each gadget consists of several assembler instructions. The last instruction is an indirect branch that serves as the connecting link between the various sequences. Traditionally, gadgets ending in a return instruction are used [44]: A return loads the next address off the stack and transfers the control to that address. The stack pointer is also incremented by one data word. Hence, the stack pointer plays an important role in ROP attacks, as it specifies which gadget will be executed next.[1]

(Fine-grained) ASLR: A widely-applied defense technique against code reuse attacks is address space layout randomization (ASLR) [22, 36, 52] which randomizes the base addresses of code and data segments. However, ASLR is vulnerable to brute force attacks [45] and memory disclosures [16]. The latter can be used to reveal important addresses at runtime. Given a leaked memory address the adversary adjusts each pointer used in the ROP payload before launching the attack. To address this problem, several fine-grained ASLR schemes have been proposed [14, 21, 26, 39, 55]. The main idea is to randomize the internal structure of an application, for example, by permuting functions [26], basic blocks [14, 55], or randomizing the location of each instruction [21]. It was believed that fine-grained randomization mitigates ROP attacks. However, a new attack strategy, Just-In-Time ROP (JIT-ROP) showed how to circumvent fine-grained ASLR with real-world exploits.

B. Just-in-Time Code Reuse

Just-in-time return-oriented programming (JIT-ROP) circumvents fine-grained ASLR by finding gadgets and generating the ROP payload at runtime using the scripting environment of the target application (e.g., a browser or document viewer). As with many real-world ROP attacks, the disclosure of a *single* runtime memory address is sufficient. However, in contrast to standard ROP attacks, JIT-ROP does not require the precise knowledge of the code part or function the memory address points to. It can use any code pointer such as a return address on the stack to instantiate the attack. Based on that leaked address, JIT-ROP discloses the content of other memory pages by recursively searching for pointers to other code pages and generates the ROP payload at runtime.

The workflow of a JIT-ROP attack is shown in Figure 1. Here, we assume that fine-grained ASLR has been applied to each executable module in the address space of the (vulnerable) application. First, the adversary exploits a memory disclosure vulnerability to retrieve the runtime address of a code pointer ❶. One of the main observations of Snow et al. [47] is that the disclosed address will reside on a 4KB-aligned memory page (Page$_0$ in Figure 1). Hence, at runtime, one can identify the start and end of Page$_0$ ❷. Using a disassembler at runtime, Page$_0$ is then disassembled on-the-fly ❸. The disassembled page provides 4KB of gadget space ❹, and more importantly, it is likely that it contains direct branch instructions to other pages, e.g., a call to Func_B ❺. Since Func_B resides on another memory page (namely Page$_1$), JIT-ROP can again determine the page start and end, and disassemble Page$_1$ ❻. This procedure is repeated as long as new direct branches pointing to yet undiscovered memory pages can be identified ❼. Using the disassembled pages, a runtime gadget finder is then used to identify useful ROP gadgets (e.g., LOAD, STORE, or an ADD ❽). Finally, the ROP payload is composed based on the discovered ROP gadgets and a high-level description of the desired functionality provided by the adversary ❾.

III. Beyond Fine-grained ASLR: Bypassing Oxymoron

Recently, several [3, 4] code randomization schemes have been proposed that aim at tackling JIT-ROP. However, at the time of writing, the first and only published approach that claims to resist JIT-ROP was the work on Oxymoron [3]. Hence, we focus our security analysis in this section on

[1] For this reason, the adversary needs to first invoke a stack pivot sequence for heap-based ROP attacks. The stack pivot sequence simply loads the address of the ROP payload into the stack pointer before the ROP attack starts executing [58].

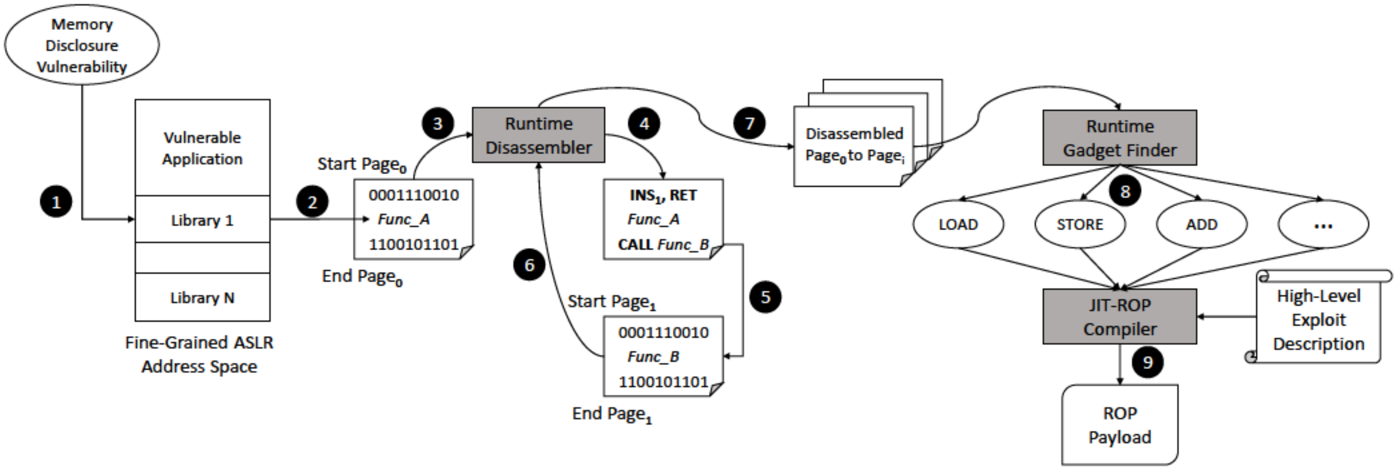


Fig. 1: Highlevel overview of a JIT-ROP attack [47].

Oxymoron, and discuss other concurrently developed defenses in related work (cf. Section VIII).

The main goal of Oxymoron is to (i) enable code sharing for randomized code, and (ii) hide code references encoded in direct branch instructions. The latter effectively prevents an adversary from discovering and disassembling new code pages (step ❺ in Figure 1), since the adversary can no longer follow a direct branch target to identify a new mapped page. Internally, Oxymoron uses a combination of page-based randomization and x86 segmentation to reach its goals. For this, Oxymoron transforms direct inter-page branches into indirect branches. The original destination addresses of all transformed branches are maintained in a special and hidden table. Specifically, the table is allocated at a random location in memory and Oxymoron assumes that the adversary cannot disclose the location and content of this table. In particular, Oxymoron forces the transformed branch instructions to address the table through a segment register which holds an index to the table. The use of a segment register creates an indirection layer that cannot be resolved by an adversary in user-mode, because the information necessary for resolving the indirection are maintained in kernel space. While Oxymoron indeed hinders JIT-ROP from discovering new code pages, we show in the following that the steps ❷ - ❺ in Figure 1 can be easily modified and bypass Oxymoron's protection. To demonstrate the effectiveness of our new technique, we developed an exploit targeting Internet Explorer 8 which bypasses Oxymoron.

A. High-level Attack Description

The main weakness of Oxymoron concerns the fact that it focuses only on hiding code pointers encoded into direct branches. However, disassembling code pages and following direct branches to new pages, is only one way of discovering addresses of new code pages. Using indirect memory disclosure, the adversary can leverage code pointers stored on the stack and heap to efficiently disclose a large number of code pages and ultimately launch a JIT-ROP attack.

Code pointers of interest are return addresses, function pointers, as well as pointers to virtual methods which are all frequently allocated on data memory. In case of programs developed in object-oriented programming languages like C++, one obvious source of information are objects which contain virtual methods. In order to invoke these virtual methods, a virtual table (vtable) is used. This table is allocated as an array containing the target addresses of all virtual methods. Since vtables are frequently used in modern applications, and since their location can be reliably determined at runtime, we exploit them in our improved JIT-ROP attack. Nevertheless, code pointers on the stack, such as return addresses, can be also leveraged in the same manner for the case the target application does not populate any vtables.

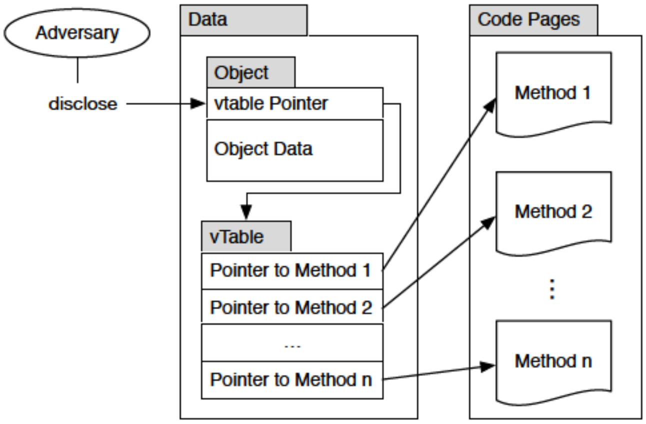


Fig. 2: Example of how disclosing a vtable pointer allows the adversary to identify valid mapped code pages.

As shown in Figure 2, the first step of the attack is to disclose the address of the so-called vtable pointer which subsequently allows the adversary to disclose the location of the vtable. Once the virtual method pointers inside the vtable are disclosed, the adversary can determine the start and end address of those pages where virtual methods reside. For a target application such as a web browser or a document viewer, it is very likely to find complex objects with numerous method pointers. A large number of method pointers increase the number of valid code pages whose page start and end the adversary can reliably infer. Given these code pages, the adversary can then perform Step ❻ to ❾ as in the original JIT-ROP attack.

In the following, we apply our ideas to a known vulnerability in Internet Explorer 8, where we assume Oxymoron's protection mechanisms to be in-place[2]. Specifically, we take an existing heap-based buffer overflow vulnerability[3] in Internet Explorer 8 on Windows 7, which is well-documented [53]. We exploit this vulnerability to validate how many code pages an adversary may identify using our above introduced techniques, and whether this code base is sufficiently large to launch a reasonable code reuse attack.

B. Exploit Implementation

As in any other code reuse attack, we require the target application to suffer from (i) a memory error (buffer overflow), and (ii) a memory disclosure vulnerability. The former is necessary to hijack the control-flow of the application, and the latter to disclose the vtable pointer which is the starting pointer to launch our attack (see Figure 2).

An additional requirement for our attack is the identification of C++ objects in Internet Explorer that populate virtual tables, i.e., contain many virtual methods. For this, we reverse-engineered C++ objects in Internet Explorer and identified several complex objects containing a large number of virtual methods (see Table I). Once we are aware of the main target C++ objects, we can pick one (or more), and write a small JavaScript program that allocates our target object on the heap.

C++ Object	Virtual Methods
CObjectElement	150
CPluginSite	150
CAnchorElement	146
CAreaElement	146
CHyperlink	146
CRichtext	144
CButtonLayout	144
...	...

TABLE I: Excerpt of C++ objects in Internet Explorer containing a large number of virtual methods

The next step is to dynamically read the vtable pointer of the target C++ object at runtime. However, this raises a challenge as ASLR randomizes code and data segments. The runtime location of the vtable pointer is not per-se predictable. However, due to the low randomization entropy of ASLR for data segments, the relative address (offset) to another memory object is in most cases predictable.

Hence, in order to exploit this circumstance, the adversary needs to allocate the target C++ object close to an *information-leak object* such as a JavaScript string. Carefully arranging objects close to each other to perform memory disclosure is commonly known as *heap feng shui* [49]. In fact, we re-use this attack technique and arrange objects using JavaScript as shown in Figure 3.

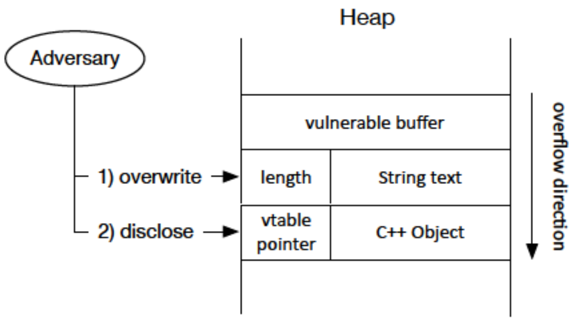


Fig. 3: Heap-Layout of our Exploit.

Specifically, we allocate via JavaScript a buffer, a string, and our target C++ object which contains many virtual methods. The string object consists of two fields, namely, the string length field holding the size of the string, and the string text itself. The memory error in Internet Explorer allows us to overflow the vulnerable buffer. As the string object is just allocated next to the vulnerable buffer, our overflow overwrites the string length field with a value of the adversary's choice. As we set the value to its maximum size (i.e., larger than the actual string length), we are able to read beyond the string boundaries. Since our C++ object (in our exploit the `CButtonLayout` object) is just allocated next to the string, we can easily disclose its vtable pointer. Afterwards, we follow the vtable pointer to disclose all methods pointers of our C++ object.

Note that Figure 3 actually contains a simplified view of our target C++ object `CButtonLayout`. By disassembling (see Figure 4) the function which creates the `CButtonLayout` object, we recognized that this C++ object contains two vtable pointers. Altogether with these two vtables we could extract 144 function pointers, and hence 74 unique code pages. In our particular exploit, the number of code pointers resp. unique pages could be increased to 322 resp. 87 pages due to the fact that the page where the two vtables of the `CButtonLayout` object reside, contains two additional vtables of other C++ objects. An adversary can always increase the number of leaked vtables by allocating more complex objects (as given in Table I) on the heap.

```
push    0FCh             ; dwBytes
push    8                ; dwFlags
push    _g_hProcessHeap ; hHeap
call    ds:HeapAlloc(x,x,x)
mov     esi, eax
[...]
mov     dword ptr [esi],
    offset const CButtonLayout::'vftable'{for 'CLayoutInfo'}
mov     dword ptr [esi+0Ch],
    offset const CButtonLayout::'vftable'{for 'CDispClient'}
```

Fig. 4: Disassembled code that creates the CButtonLayout object

The 87 leaked code pages give us access to a large code base (348 KB) for a code reuse attack. Hence, the next attack step involves gadget search on the 87 leaked code pages. For our proof-of-concept attack, we identified all

[2] Note that Oxymoron's source code is not public. Hence, we simply assume its protection is active.

[3] CVE-2012-1876

gadget types (load, store, add) necessary to launch a practical return-oriented programming attack; including a stack pivot gadget [58]. One important gadget is a system call gadget to allow interaction with the underlying operating system. The original JIT-ROP attack leverages for the dynamic loader functions `LoadLibrary()` and `GetProcAddress()` allowing an adversary to invoke any system function of his choice. However, when the addresses of these two critical functions are not leaked (as it is the case in our exploit), we need to search for an alternative way. We tackle this problem by invoking system calls directly. On Windows 32 bit, this can be done by loading (i) the system call number into the `eax` register, (ii) a pointer to the function arguments into `edx`, and (iii) invoking a `syscall` instruction on our leaked pages. At this point, we are able to compile any return-oriented programming payload as our leaked code pages contain all the basic gadget types.

Specifically, we constructed an exploit that invokes the `NtProtectVirtualMemory` system call to mark a memory page where we allocated our shellcode as executable. We use a simple shellcode, generated by Metasploit [33] that executes the `WinExec()` system function to start the Windows calculator to prove arbitrary code execution.

The last step of our attack is to hijack the execution-flow of Internet Explorer to invoke our gadget chain. We can do that simply by exploiting the buffer overflow error once again. In contrast to the first overflow, where we only overwrote the string length field (see Figure 3), we overwrite this time the vtable pointer of our target C++ object, and inject a fake vtable that contains a pointer to our first gadget. Afterwards, we call a virtual method of the target C++ object which redirects the control-flow to our gadget chain (as we manipulated the vtable pointer).

Lessons learned: In summary, our attack bypasses Oxymoron as it discovers valid mapped code pages based on code pointers allocated in data structures (specifically, virtual method pointers). As Oxymoron only protects code pointers encoded in branch instruction on code segments, it cannot protect against our improved JIT-ROP attack. In order to defend against this attack, one also needs to protect code pointers allocated in data structures. Note that our attack is general enough to be applied to any other memory-related vulnerability in Internet Explorer, simply due to the fact that Internet Explorer contains many complex C++ objects with many virtual methods (see Table I).

IV. Isomeron: Designing Code Randomization resilient to (JIT) ROP

A. Design Decisions

As the first step in designing a diversifier secure against (JIT) code reuse attacks we evaluated related approaches that could serve our purpose. One possible solution is to apply constant re-randomization as proposed by Giuffrida et al. [18]. However, the adversary could exploit the (small) time frame between the subsequent randomization to launch the attack. Another approach is to combine instruction-set randomization (ISR) [25] and fine-grained randomization. ISR encrypts the application code using a secret key. It aptly prevents an adversary from disassembling code at runtime – a crucial step in a just-in-time ROP attack. However, the original ISR proposal [25] uses XOR which has been shown to be vulnerable to known-plaintext attacks [50, 56]. Hence, we replaced XOR by the AES encryption scheme supported by Intel CPU AES instructions. Unfortunately, this solution turned out to be impractical, primarily due to the fact that repeated cryptographic operations induce an unacceptable performance degradation.

Based on the learned lessons we decided for a new diversifier approach that combines fine-grained randomization of the program code with the execution path randomization of the same code. This construction breaks the gadget chain in both ROP and JIT-ROP attacks. We call our runtime diversifier *Isomeron*. Before going into the details of Isomeron, we first explain the underlying assumptions, threat model, and security objectives.

B. Assumptions

Non-Executable Memory: We assume that all memory pages are either marked as executable or writable, thus preventing code injection attacks. This is a reasonable assumption, as W$\oplus$X is typically supported on every modern operating system and enabled by default.

Fine-Grained ASLR: We assume that the underlying system deploys fine-grained ASLR. Hence, the ROP gadgets contained in the original code image either (i) reside at a different offset, (ii) are eliminated by replacing instructions with an equivalent instruction, or (iii) are broken due to instruction reordering or register replacement [14, 21, 39, 55]. The diversifier should ensure that gadgets with the same offset in both binaries are semantically different.

Trust in the diversifier: We assume that the adversary cannot tamper with Isomeron. We also assume the availability of a trusted source of randomness. Despite these assumptions we will elaborate in Section VI on deploying techniques to protect Isomeron.

C. Adversary Model

We consider a strong adversary model that is typical for advanced attacks, such as JIT-ROP:

Exploiting memory vulnerabilities: The adversary has knowledge of a vulnerability in the software running on the platform, allowing the adversary to instantiate a runtime attack.[4]

Full memory disclosure: The adversary has access to all code pages mapped into the address space of an application.[5] Full memory disclosure also implies that the adversary can circumvent fine-grained ASLR protection schemes (which we already assumed to be deployed on the target system).

Brute forcing: The adversary has a limited number of attempts for the attack. We assume victims would not re-open a webpage or document after it has crashed multiple times.

[4] Reasonable assumption since the NIST vulnerability database shows 760 CVE entries in the buffer error category for 2013.

[5] In practice, a JIT-ROP adversary can only access pages whose addresses she disclosed.

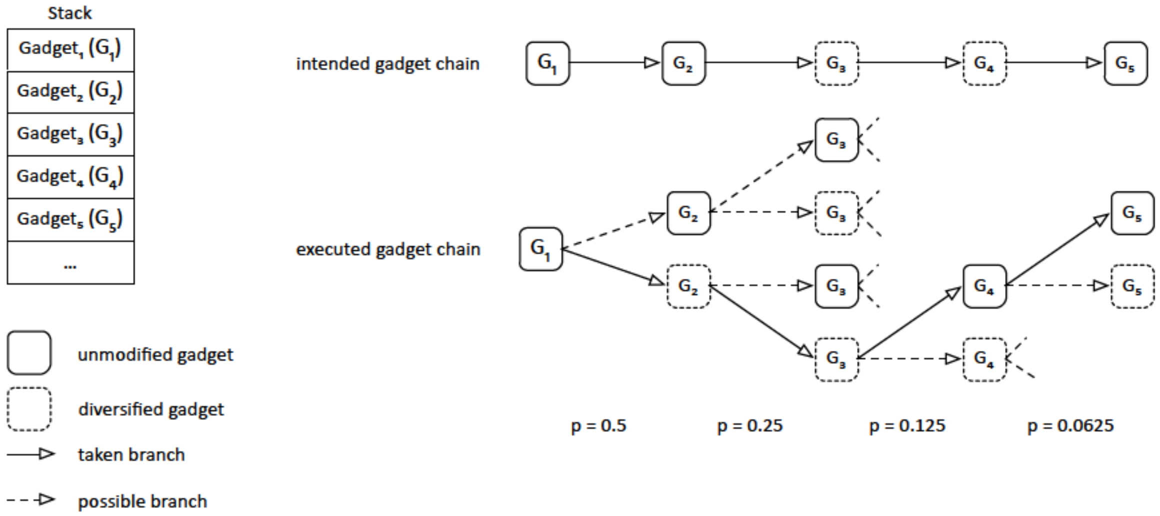


Fig. 5: High-level idea of Isomeron's execution path randomization: The control flow continues either in the unmodified or in the diversified program copy, based on the random decisions of Isomeron.

D. Security Objectives

Our main objective is to mitigate traditional and JIT-ROP attacks. ROP attacks are conducted through different types of indirect branches as described in Section II-A. In the following we consider these attack vectors and identify the required protection mechanisms.

Protection against traditional ROP: The ROP adversary analyzes the code and constructs the gadget chain prior to the execution of the targeted (vulnerable) application. Hence, we require a mechanism that changes the addresses of the gadgets and consequently breaks the gadget chain. As described in Section II-A, any (fine-grained) randomization, applied before the application is executed, is suitable.

Protection against JIT-ROP: In contrast to traditional ROP, the JIT-ROP adversary has knowledge of the memory layout and the content at runtime. We require a mechanism to counter memory disclosure vulnerabilities as explained in Section IV-E.

Protection against ret-to-libc and jump-oriented programming: ret-to-libc [48] may be considered as an instantiation of ROP. Given the assumption of memory leakage, randomization schemes are unable to defeat ret-to-libc [3, 21, 39, 55], which also seems to be a hard problem in general. Nevertheless, we require in our solution techniques that drastically reduce the success probability of ret-to-libc. These techniques also apply to jump-oriented programming [8].

Protection of diversifier: Although we assume that our runtime diversifier Isomeron is trusted, we explore mechanisms to protect Isomeron from being compromised in Section VI.

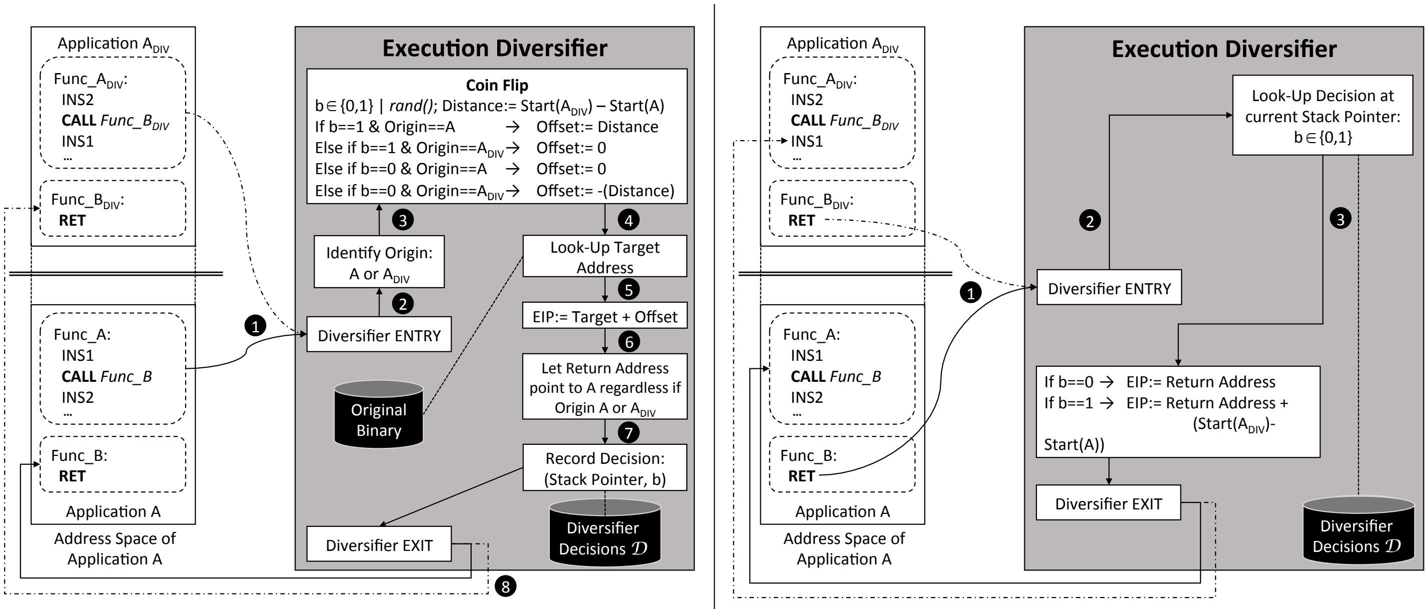
E. High-level Idea of our Solution

Just-in-time code reuse relies on two distinct steps: disclosing memory to construct a payload as shown in Figure 1, and exploiting a vulnerability to execute the generated payload. Thus, we need to ensure that the addresses of the gadgets in the gadget chain change after the chain is built. This will result in an undefined behavior of the payload (likely leading to a crash of the program). We simultaneously load two copies of the program code in one program's virtual address space. One copy is the original application code A, while the other is *diversified* A_{div} using any fine-grained ASLR. While the program is executing, we continuously flip a coin to decide which copy of the program should be executed next. Since the gadget sets are completely different in each program copy, and it is not predictable which copy will execute at the time of exploitation, the adversary is unable to reliably construct a payload, even with full knowledge of all memory contents. This process is illustrated in Figure 5. Note that the adversary in this example is aware of diversified gadgets. However, the adversary cannot predict whether the unmodified or the diversified version of a gadget will be executed. Hence, the probability of guessing the correct sequence decreases exponentially with the length of the gadget chain.

F. Architecture of Isomeron

While the high-level idea is conceptually simple, we encountered a number of practical problems that posed significant challenges on constructing our defense that we describe in the following. Figure 6 illustrates our diversification framework.

Step 1: *Program twinning*. For cloning a program image within a single virtual address space, we apply dynamic binary instrumentation. Similar to common instrumentation frameworks, we instrument code on the granularity of basic blocks (BBLs). However, instead of emitting a single BBL into one instrumentation cache, our framework can emit multiple (diversified) copies into multiple instrumentation caches (A and A_{div}). This specific feature of program twinning motivated us to develop our own instrumentation framework, since currently available solutions do not per-se support code duplication [6, 31].


Application A$_{DIV}$
Func_A$_{DIV}$:
INS2
CALL Func_B$_{DIV}$
INS1
...
Func_B$_{DIV}$:
RET
Func_A:
INS1
CALL Func_B
INS2
...
Func_B:
RET
Application A
Address Space of Application A
Execution Diversifier
Coin Flip
b∈{0,1} | rand(); Distance:= Start(A$_{DIV}$) – Start(A)
If b==1 & Origin==A → Offset:= Distance
Else if b==1 & Origin==A$_{DIV}$ → Offset:= 0
Else if b==0 & Origin==A → Offset:= 0
Else if b==0 & Origin==A$_{DIV}$ → Offset:= -(Distance)
Identify Origin: A or A$_{DIV}$
Diversifier ENTRY
Original Binary
Look-Up Target Address
EIP:= Target + Offset
Let Return Address point to A regardless if Origin A or A$_{DIV}$
Record Decision: (Stack Pointer, b)
Diversifier EXIT
Diversifier Decisions $\mathcal{D}$
❶
❷
❸
❹
❺
❻
❼
❽
Look-Up Decision at current Stack Pointer: b∈{0,1}
If b==0 → EIP:= Return Address
If b==1 → EIP:= Return Address + (Start(A$_{DIV}$)-Start(A))

(a) Function calls

(b) Function returns

Fig. 6: Details of function call and return hooking of Isomeron.

Step 2: *Twin diversification (Isomer)*. In the offline or load-time phase, we apply fine-grained ASLR to the executable modules of an application. The level of fine-grained ASLR is configurable, but we require and ensure that each possible ROP sequence and gadget is placed at a different address. In other words, a ROP sequence should never reside at the same offset in the original and diversified version. This requirement is fulfilled by all of the proposed fine-grained ASLR solutions [14, 21, 26, 39, 55]. Depending on the design of the chosen fine-grained ASLR solution, the diversification is performed once; either within a static offline phase (as done in [21, 26, 39, 55]), or completely at load-time of the application [14].

Step 3: *Coin-flip instrumentation*. We perform the execution randomization at the granularity of function calls. Our scheme randomly flips a coin, and based on the outcome, it continues the execution in either the diversified or the original application. This random decision is made whenever a function call occurs. Our execution diversifier $\mathcal{D}$ ensures that the function is completely executed either from the original A or diversified code image A_{div}. The rationale behind performing the random decision on function level granularity is that we can only preserve the original semantics of the application when a function is entirely executed from either the original or the diversified address space. Note that fine-grained ASLR is only performed once (either offline or at load-time), while the execution path randomization is performed throughout the entire program execution, each time a function is called.

Figure 6(a) and 6(b) show the instrumentation framework for function calls and function returns. Subsequently, we describe the specifics of each. For brevity, we use an example that consists of only two functions: *Func_A()* and *Func_B()*. The latter function only contains the x86 return instruction (`RET`), while the former one contains two instructions (`INS1` and `INS2`), and a function call to *Func_B()*. The only code diversification we apply in this example is that `INS1` and `INS2` are exchanged.

1) Instrumentation of direct function calls: First, we need to ensure that we take control over the execution flow when a function call occurs. For this, we perform binary rewriting on the code, before it is executed. Specifically, we overwrite each function call with a call to the execution diversifier $\mathcal{D}$. We describe the implementation details of our rewriting approach in Section V. For the moment we assume an in-memory rewriter allowing us to hook into function calls.

For our running example, the call to *Func_B()* will either be initiated in the original or diversified version of *Func_A()*. Since the call has been instrumented in any case, the control-flow will hand-over control to the diversifier $\mathcal{D}$ (❶). Moreover, since the (instrumented) function call will automatically push the return address onto the stack, $\mathcal{D}$ can easily identify from where the function has been invoked. Note that the memory layout of the running application is known to $\mathcal{D}$ (❷).

Next, we perform a coin flip (❸) to decide which version of `Func_B()` is going to be executed. At this point, $\mathcal{D}$ also determines the memory offset between the diversified and original images. This offset is used to quickly calculate[6] the function address during the execution path randomization process. Based on the origin and the outcome of the coin toss, we calculate the *offset* to be added to the instruction pointer. In general, the offset is zero if execution should continue on the image from where the function call originates. If the program image is switched from original to diversified or vice-versa, the offset will be the distance between the two program images.

In steps ❹ and ❺, we retrieve the original target address

[6] This calculation needs only to be performed once and can be retrieved in future coin flip rounds.

of the call (using the original binary), and add the offset (calculated in ❸) to determine the new value of the instruction pointer (on x86: `eip`). In addition, we ensure that the return address on the stack always points to the original program modules (❻). Otherwise, an adversary could determine which functions are currently executing by inspecting the stack. We also record each random decision of the diversifier $\mathcal{D}$ to ensure that function returns are returning to their caller in the correct program version. This information is only accessible to $\mathcal{D}$ (❼). Finally, $\mathcal{D}$ redirects the control-flow to *Func_B()* (❽).

2) Instrumentation of function returns: Function returns are instrumented similarly to function calls. The diversifier $\mathcal{D}$ takes over the control whenever the program issues a return instruction. Next, the current return address is read from the stack, and used to determine the correct origin, which is adjusted if necessary. Recall that the return address always points to the original image. Since previous decisions are unknown to the adversary, she can only guess whether the return address is adjusted or not. This knowledge is crucial for an adversary; in our example, either `INS1` or `INS2` will be executed next.

3) Instrumentation of indirect jumps and calls: Indirect branches are handled similarly to direct branches. The difference is that indirect branches can have multiple branch targets. Hence, we calculate the destination address at runtime and check if the target address is included in the relocation information. This limits potential misuses. We discuss the details in Section VI. The relocation information are used by Windows to implement ASLR and are therefore almost always available[7].

V. Implementation of Isomeron

Our design, as presented in Section IV, can be implemented in two ways: As a compiler extension or through binary instrumentation. While the former has advantages in terms of performance and completeness, the latter is compatible with legacy applications. For our proof of concept, we choose to use dynamic binary instrumentation.

A. Dynamic Binary Instrumentation

Dynamic binary instrumentation (DBI) [37] can be seen as a form of process virtualization with the goal to maintain control over the executed code. To achieve this goal, all control transferring instructions are modified (translated) such that the dynamic binary instrumentation software controls which instruction is executed next. Dynamic binary instrumentation has been used for runtime monitoring [35, 54] and policy enforcement [11, 13]. It fetches the code on a basic block granularity and translates the instructions inside a basic block to meet the desired functionality. The translated instructions are emitted in a *basic block cache* (BBCache), which is an executable area of the memory that contains all translated basic blocks. Translation in this context means that the framework modifies the instructions according to the purpose of the intended instrumentation. At the end of a translated basic block an exit stub is emitted. The exit stub saves the current execution context and transfers the control to the instrumentation framework, which contains the runtime information needed to calculate the address of the next basic block.

In contrast to static binary instrumentation, dynamic binary instrumentation has access to information which are calculated at runtime (e.g., pointers). We decided to use the dynamic approach, because it has several advantages with respect to our design goals: first, it covers all (in)direct branches as it instruments the code right before it is executed. This also covers just-in-time generated code. Second, our solution requires the insertion of new instructions which inevitably changes the location of the original instructions. Hence, all references to a moved instruction must be adjusted throughout the entire binary. The dynamic approach allows us to keep track of these changes and adjust references accordingly. Lastly, we require our solution to be compatible with legacy applications. Thus, we cannot assume access to source code or debugging symbols which are required to perform a reasonable static analysis.

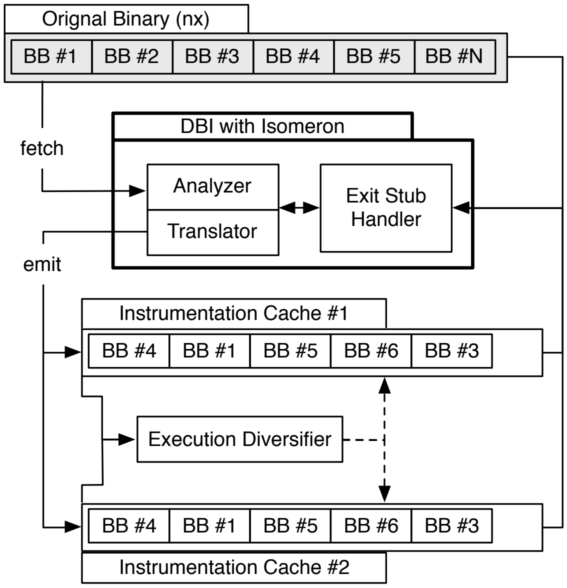


Fig. 7: Components used in our dynamic binary instrumentation framework.

The informed reader might realize that there are several well-known instrumentation frameworks (e.g., DynamoRIO [6] and PIN [31]) which we could have used. Unfortunately, these frameworks are not suitable for our purposes as we require the ability to emit (generate) differently instrumented copies of the same code. While DynamoRIO's sources are available, the large optimized code base is not practical for drastic architectural changes, such as introducing a second code cache. Other open source frameworks are only available for Linux [41, 51]. Next, we highlight the challenges we encountered, but before doing so, we describe the start-up process and explain how new basic blocks are discovered.

[7] In order to avoid emission of relocation information, a developer would need to explicitly disable ASLR support when compiling with Microsoft's standard compilers.

B. Implementation of our Dynamic Binary Instrumentation Framework

The design of dynamic binary instrumentation frameworks is similar across different implementations and illustrated in Figure 7. In the following we will explain how our framework is instantiated. We fetch basic blocks from the original binary (marked as non-executable), analyze, translate and emit them into a code cache. One major difference to existing instrumentation frameworks is our translator emits an additional instrumented, diversified basic block into a second code cache. Further the execution path diversifier switches the execution between both code caches.

1) Setup: We implemented our instrumentation framework as a dynamic link library (DLL). To instrument a program, we inject our library into the target process. We created a loader that starts the target application as a *suspended process*. This means the Windows loader stops the process creation right before the execution of the main application code is started. This gives us the opportunity to inject our library and take control over the process, i.e., to start the instrumentation of the first basic block and to initialize necessary code/data structures.

2) Initialization of code and data: One of the main challenges of dynamic binary instrumentation is thread support. Each thread needs its own memory region to store information, e.g., during the translation or for the applied instrumentation (see Figure 6(a)). Further, each thread must be able to access its memory region quickly due to performance reasons. There are different strategies to implement this. One obvious strategy is to use the `Thread Local Storage`, which is provided by the operating system. However, dynamic binary instrumentation should minimize the interaction between the framework and the operating system at runtime. Another strategy is to store a pointer to the thread's private memory in one of the registers. This is called register stealing and works well on architectures with many general-purpose registers, but since x86 provides only a few of these registers, register stealing has a negative impact on performance [6]. For our implementation, we chose to access the thread's private memory through hardcoded addresses. Hence, if a new thread is started we first create a copy of each function that requires access to the thread's private memory area and then adjust the reference of every instruction which accesses the private memory.

3) Basic Block translation: As mentioned above, our loader stops the process right before the first basic block is executed. Hence, we start our instrumentation with this particular basic block. To instrument a basic block, we first have to disassemble and analyze its code. While creating the correct disassembly for an entire binary is error-prone, disassembling a basic block is not. For our implementation we chose `libdasm` [30], a lightweight disassembler that requires no external libraries. Most instructions are copied directly and without any modifications into the instrumentation cache. However, there are three cases at which we apply modifications during the translation.

Control flow instructions: For simplicity, unless a basic block is already translated, we emit an exit stub on each control flow altering instruction (calls/jumps/returns). It is the responsibility of the exit stub to save the state of the current execution and transfer the execution to our instrumentation framework, which then can use the saved state to determine the next basic block. After the translation of the next basic block, the exit stub is replaced with code that transfers control to the execution diversifier, in the case of calls and returns. Otherwise, it is replaced with a direct jump to the next basic block within the current instrumentation cache.

Unaligned gadgets: Isomeron needs to ensure that its execution diversifier is correctly executed. Since it instruments only intended instructions, we need to prevent the adversary from diverting the execution flow to instructions that are not instrumented. To handle unaligned instructions, we search for instructions inside a basic block that contain byte values that could be exploited by an adversary as an unaligned gadget (e.g., a `C3` byte inside a `MOV` instruction, which could be interpreted as a return instruction). If such an instruction is found, we insert alignment instructions which ensure that these bytes can only be interpreted as part of the intended instruction. Another way to avoid unaligned gadgets is to enforce alignment for branch destination addresses [32].

Path and Code diversification: The main focus of this paper is the implementation of execution path randomization. Therefore, we redirect call and return instructions to our execution diversifier which we explain subsequently. Our framework allows to use code randomization as a black box. Hence, the translator applies randomization of a proper fine-grained randomization scheme (c.f. Section IV-F) before emitting the instrumented code into the diversified code cache.

4) Execution diversifier: Our execution diversifier implementation follows the description given in Section IV. For efficiency, we implemented separate handlers for direct/indirect calls and returns. In order to preserve the semantics of the function, jumps are never randomized and hence, are not target of the execution diversifier. Nevertheless, we apply certain limitations to indirect jumps as discussed in Section IV and VII. For efficiency the source for our random decision is a pre-allocated area of memory initialized with random values. This source can be re-randomized in certain intervals to prevent an adversary from using side-channels to disclose any of the random bits. The coin flip results are saved on an isolated memory location which is allocated for each thread.

VI. Security Considerations

We elaborate on how our implementation can fulfill the security objectives of Section IV-D.

The security of Isomeron is based on the uncertainty for the adversary to predict the outcome of the random decisions taken by the diversifier, i.e., whether the execution takes place in the original or in the diversified program copy. In fact, this means that the adversary cannot anticipate which gadget chain and instructions are executed after the control-flow has been hijacked. For example, a value that is intended to be stored into the register A could be loaded into register B instead. Hence, guessing the wrong gadget leads to the wrong exploit state, i.e., register A contains an undesired value.

This even holds for special gadget pairs (G_i, G_{nop}), where at a given offset one program copy performs the gadget (intended by the adversary) G_i, and the other one simply executes a `nop` gadget G_{nop}. While the occurrence of such gadget pairs is potentially possible, they still lead to the false exploit state, because the adversary does not know whether G_i or G_{nop} is going to be executed. To tackle this problem, an adversary could increase the success probability of G_i to be executed by invoking the gadget pair multiple times in a row. However, the adversary still cannot know how often G_i is actually executed. Hence, this limits the gadget space for G_i to gadgets which do not modify their own input value. For instance, a gadget which adds a value to a register and saves result in the same register cannot be used, because the adversary cannot predict the final value in the register. This limitation excludes many traditional gadgets like those using the `pop` instruction to load a value into a register (as the adversary cannot predict the stack state). Although it remains open whether it is possible to create a practical attack payload under these constraints, a code diversifier can mitigate the threat of `nop`-gadget pairs by ensuring that all gadget pairs have some undesirable side-effect, which we consider for future work.

(JIT) ROP: As mentioned in Section IV-C, we assume that the adversary is capable of disclosing most of the address space of an application and assembling her payload using gadgets from the original and diversified copy at runtime. However, before the adversary diverts the control flow to the ROP chain, this chain must contain all addresses the adversary intends to execute. She cannot modify the ROP chain, after the control flow has been hijacked. This principle holds for traditional ROP as well as JIT-ROP. Our approach is to hinder the adversary's ability to successfully execute a ROP gadget chain. Recall that each image will contain different ROP gadgets, as fine-grained ASLR will eliminate or break the original ROP gadgets in the diversified image. Since each pointer in the ROP payload must have a fixed address, the adversary can either select the gadget from the original or from the diversified image. However, due to execution path randomization, the adversary has a chance of $p = 0.5$ to guess correctly (for *each* gadget) that the execution will transfer to the intended gadget and the adversary's chances of successfully completing a ROP chain will exponentially drop with the length of the ROP chain. As stated in Section IV-B, the effectiveness of our approach relies on the integrity of the instrumentation framework.

We also note that our approach heavily depends on instrumenting program code. Yet, the fact that the x86 architecture allows for variable-length instruction raises further technical challenges, because a single byte stream can be interpreted in different ways, depending on the location where the decoding begins. Thus, one subset of instructions in a program are the "intended" instructions authored by the software developer (aided by the compiler), while instructions decoded from any other starting location are "unintended" or "unaligned". It has been shown that one can construct construct Turing-complete payloads using only unaligned instructions [44]. We apply a simple countermeasure to eliminate the problem of unaligned gadgets all-together. We adopt the idea suggested by Onarlioglu et al. [38] whereby unintended gadgets are eliminated by inserting alignment instructions, such as `NOP`s before instructions which contain potentially helpful bytes for a ROP payload. The resulting effect is that the byte-stream of two instructions which might contain an unaligned instruction will be separated eliminating the ROP gadget. We adopt the compiler-based techniques of Onarlioglu et al. [38], but apply them at runtime instead.

Ret-to-libc: Our adversary model assumes that the adversary is aware of all functions within the process space. Like other approaches [3, 21, 39, 55], we cannot completely prevent ret-to-libc attacks. Randomizing the execution flow at a function-granularity level does not affect ret-to-libc, because, as stated in Section IV-F, the semantics of a function does not change. However, we limit the number of possible ret-to-libc targets using the application's relocation information. Function addresses that are included in the relocation information might be used in a ret-into-libc attack, because they are legitimate indirect jump targets.

Jump-oriented programming: To mitigate this attack technique, we limit the potential jump target addresses to those included in the relocation information. By analyzing the SPEC tools, we discovered, that on average 92% of the indirect jumps are performed using jump tables. Using the relocation information, we can reliably identify the tables and the targets for individual jumps and limit these jumps to the identified benign targets. This leaves on average 8% of all indirect jumps available for jump-oriented programming.

Disclosure of execution diversifier data: Accessing the data of the execution diversifier can determine the correct location of the intended gadgets. In our current implementation this information is part of the trusted computing base as described in Section IV-B, not accessible to the adversary. Technically there are different possibilities to realize this, either using segmentation (e.g., as done in [3]) or by using SGX.

Return to unaligned instructions: In our implementation, return instructions can only change the instruction pointer to a previously instrumented basic block. It is not possible to return to a not instrumented basic block and trick the instrumentation framework into assuming a new basic block was discovered. An instrumented call is always followed by a direct jump, either to an exit stub, in case the subsequent basic block is not instrumented yet, or to the instrumented basic block. Hence, a return can safely return to the same BBL the call originated from.

VII. Evaluation

In this section, we evaluate Isomeron's efficiency as well as the effectiveness against code reuse attacks, in particular against JIT-ROP. To evaluate its effectiveness, we make use of a vulnerable test application that contains two representative memory-related vulnerabilities. The first one is a format string vulnerability that allows an adversary to leak memory content from the stack. The second is a stack-based buffer overflow vulnerability that allows to overwrite a function's return address and, hence, to hijack the execution flow of the application. These two vulnerabilities can be exploited to first bypass ASLR, and then launch a ROP attack. Note that similar vulnerabilities are continuously discovered in real-world applications. Moreover, while our proof-of-concept exploit uses stack-based vulnerabilities, our solution equally applies to exploits that leverage heap-based vulnerabilities.

For evaluation purposes, we consider a traditional ROP attack as successful if the adversary is able to call a ROP gadget that writes the value 1 into the `eax` register. For our experiment, we applied in-place randomization [39]. As expected, the attack against our vulnerable application fails, because the selected gadget was broken by the applied randomization scheme. In a second attempt, we apply `nop` insertion which shifts the location of all instructions. Again, the attack failed. The security in this experiment relies on the secrecy of the chosen randomization scheme.

For our proof-of-concept implementation we only use a randomization offset of a single byte. As we will argue in the following, this very low entropy is already sufficient to reduce the attack success rate to 50% even if the adversary can disclose memory content (using principles of JIT-ROP).

In a JIT-ROP attack, the adversary is (potentially) aware of any byte contained in the address space of the application. To demonstrate the effectiveness of Isomeron against a JIT-ROP adversary, we again deploy our vulnerable application. We let Isomeron randomize the application code and execution path. For this, we maintain two images of an application in memory (one unmodified, and a diversified one where instructions are shifted by one byte). We assume that the adversary knows the address of the desired gadget in both copies (`0x001E8EB3` and `0x19F8EB4` in Figure 8).

Before a ROP chain is executed, the adversary has to decide whether she chooses to execute the gadget in the original (A) or diversified copy (A_{div}). The decision needs to be made before the ROP attack is launched, since the address of the gadget needs to be written onto the stack. In the example, the adversary chooses the original gadget in A (`0x001E8EB3`). Since the sequence of coin flips applied to function calls is unknown to the adversary, there is only a chance of $p = 0.5$ that the intended gadget is executed. In this example the adversary predicted the wrong location. Hence, Isomeron modifies the return address by adding the offset between both images. Since randomization is applied, the instruction pointer is not set to the desired ROP gadget, but to the last byte of the previous instruction. In this particular case the privileged instruction `IN` is executed, which leads to an immediate crash of the process.

As noted above, this simplified attack has a success rate of 50% which provides an ample opportunity for an adversary to succeed. However, real-world exploits typically involve invoking several gadgets, each of which have a probability of $p = 0.5$ of being successfully executed. According to Cheng et al. [10] the shortest gadget chain Q [43], an automated ROP gadget compiler, can generate consists of thirteen gadgets. Hence, the probability for successful execution of a gadget chain generated by Q is lower or equal to $p = 2^{-13} = 0.000122$. Snow et al. [47] successfully exploited a vulnerability (`CVE-2010-1876`) targeting the Internet Explorer with a ROP payload that consists of only six gadgets, which would equate to a better, but still low, success probability of $p = 2^{-6} = 0.0156$.

Gadgets	**PL$_1$**	**PL$_2$**
unique found	102	102
used	16	31
unique used	8	8
diversified (unique; used)	5; 12	5; 25
probability of success	p = 0.00024	p = 0.00000003

TABLE II: Analyzed gadgets found by JIT-ROP.

We now turn our attention to the analysis of ROP gadgets in two JIT-ROP payloads (PL$_1$ and PL$_2$) [47]. The results are summarized in Tabel II. PL$_1$ represents a very simple payload that just opens the Windows calculator. JIT-ROP discovered in total 102 unique gadgets. The generated payload consists in total of 16 gadgets. Eight of the used gadgets are unique and affected by our diversification. Note that our proof-of-concept implementation of nop insertion does not affect all gadgets, e.g., gadgets which are comprised of entire BBLs. However, we only chose nop insertion due to its simplicity, and one can simply use in-place instruction randomization as performed in [39] to increase the randomization entropy. Since these gadgets were used multiple times, the probability that the ROP chain gets executed correctly is $p = 2^{-12} \simeq 0.00024$. PL$_2$ represents an average size payload. It also starts the Windows calculator but additionally performs a clean exit of the vulnerable application. Our diversification affects 25 out of 31 used gadgets, reducing the probability of success to $p = 2^{-25} \simeq 0.00000003$.

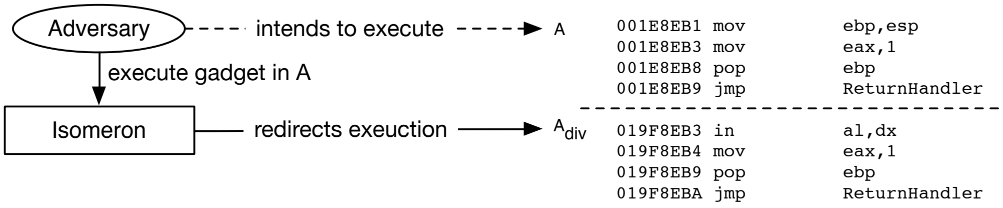
We reliably identify jump tables using the relocation information of the binary during the translation process. We limit the target addresses of an indirect jump to the legitimate addresses that are listed in the corresponding jump table. We evaluated the effectiveness for the SPEC tools using IDA Pro. In Table III, we list the percentage of indirect jumps that use a jump table to determine their destination address. Limiting indirect jumps to their benign target addresses decreases the number of potentially useable indirect jumps on average by 92.07%. Note that the remaining 8% might not even be suitable for jump-oriented programming, because indirect jump gadgets must fulfill certain requirements [8].

SPECINT	**table jumps**	**SPECFP**	**table jumps**
400	96.51	433	91.90
401	93.85	444	84.10
403	99.47	447	89.55
429	93.55	450	92.13
445	93.55	453	97.20
456	94.45	470	90.91
458	95.3		
464	93.26		
471	83.56		
473	83.77		

TABLE III: Percentage of indirect jumps for each SPEC tool that use a jump table.

Runtime performance: To be comparable to other solutions [3, 13, 14, 18, 21, 55], we measured the the CPU time overhead of Isomeron, by applying it to SPEC CPU2006. We conducted our performance tests on a 3.1 GHz i5-2400 machine with 4 GB RAM running Windows 8.1 32 bit. The SPEC benchmarking tools as well as Isomeron were compiled with the Microsoft compiler version 16.00.30319.01 with full optimization enabled. For the SPEC tools we selected the default input for reference measurements, ran each tool three times, and selected the median for our statistic.

Figure 9 shows the results of our performance evaluation using the SPEC benchmarking tools. We measured the overhead of PIN and our dynamic binary instrumentation framework (DBI) without any activated instrumentation tools, and our DBI with execution path randomization enabled.



```
001E8EB1 mov        ebp,esp
001E8EB3 mov        eax,1
001E8EB8 pop        ebp
001E8EB9 jmp        ReturnHandler
-----------------------------
019F8EB3 in         al,dx
019F8EB4 mov        eax,1
019F8EB9 pop        ebp
019F8EBA jmp        ReturnHandler
```

Fig. 8: Targeted (upper) and executed gadget (lower).

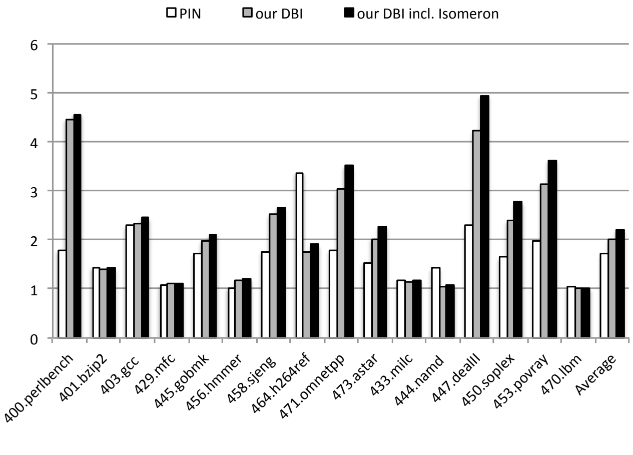


Fig. 9: CPU Runtime Overhead for SPEC CPU2006.

Compared to our framework, PIN induced less overhead for most benchmarks, because it implements mature optimizations techniques. However, in some cases (e.g., 429.mfc) these optimizations are not triggered and the overhead induced by our framework is comparable to the overhead of PIN. Note that as discussed in Section V PIN cannot fulfill the requirements imposed by Isomeron. The activation of execution path randomization in our instrumentation framework induces an additional overhead which is on average 19%.

The noticeable variation of overhead between individual measurements is the result of our instrumentation framework, since specifically in our current implementation most indirect branches require a full context switch to the framework and then back to the application, except from indirect jumps that are computed through jump tables. On the other hand, dynamic binary instrumentation inherently introduces a non-negligible performance overhead. As discussed in Section V we decided to use dynamic binary instrumentation to be compatible to legacy application.

Limitations: The implementation of our instrumentation framework focuses on features that are crucial to prove our concept, namely multiple copies and Windows support. Therefore, our instrumentation framework is not as mature as pure instrumentation frameworks [6, 31] in terms of performance and compatibility. Even so, it demonstrates the feasibility of Isomeron, where the average overhead of execution path randomization is 19% in our empirical evaluations. This degradation in performance is somewhat expected, given that our program shepherding renders certain CPU optimizations (e.g., branch prediction) ineffective. However, in contrast to most control-flow integrity solutions [1, 57], which require static analysis, dynamic binary instrumentation can also handle dynamically generated code.

Except Oxymoron [3], fine-grained randomization schemes [14, 21, 39, 55] do not allow code sharing among processes, because they randomize the program code (for security reasons) differently for every process. Our current implementation of Isomeron also does not allow code sharing, because we use dynamic binary instrumentation. However, in contrast to other fine-grained randomization schemes we assume that the original and diversified code is known to the adversary. Hence, our security is not weakened if the same code randomization is applied to a library that is shared among processes. The best way to achieve this is to use a compiler.

The components of Isomeron must be protected against code and data tampering (cf. Section IV-B). Both requirements can be fulfilled by applying segmentation. This x86 legacy feature allows for dividing the memory into different segments, where one segment is more privileged than the other. Hence, embedding Isomeron into the more privileged segment ensures that it cannot be accessed or modified by the adversary, while Isomeron can still instrument the target application. One disadvantage of segmentation is that it is only available for 32 bit applications. However, Intel's upcoming *Software Guard Extension* (SGX) [23] can provide a protected environment within a process for 32 and 64 bit applications.

VIII. Related Work

In general, ROP mitigation techniques can be categorized into two classes. The first class tries to passively prevent ROP by removing gadgets from the binary [29, 38] or randomizing critical memory areas. The latter is implemented in modern operating systems using ASLR. However, as we mentioned before, ASLR is vulnerable to memory disclosure and brute-force attacks (see Section II-A). To improve simple ASLR, recent research has focused on increasing the randomization entropy through fine-grained ASLR [28]. The proposed randomization techniques range from basic block [14, 55] and function permutation [26] to randomization at instruction level granularity [21]. While the aforementioned solutions are based on binary instrumentation, one can also apply source code transformation to produce binaries that apply fine-grained randomization at load-time [5]. However, all these solutions have been shown to be vulnerable to JIT-ROP attacks that dynamically generate ROP payloads at runtime.

One approach to tackle the weaknesses of fine-grained ASLR is to periodically re-randomize program code [18]. However, the approach has only been validated for a microkernel and only to software running in kernel mode. We belive that the same re-randomization techniques will likely induce significant performance overhead for modern user-level programs. Another defensive strategy is to perform explicit checks based on predefined policies. A prominent technique of this class is control-flow integrity (CFI). The main idea is to derive an application's control-flow graph (CFG) prior to execution and validate at runtime whether an application follows a valid path in the CFG [1]. Proposed solutions range from label checking [1], maintaining a shadow stack [13, 46] over light-weight techniques to ensure that the return address points to an instruction directly following after a call instruction [57]. An even more coarse-grained approach of CFI is to enforce ROP chain detection during critical function calls [40]. While the performance was indeed improved, the relaxation of the policies enlarged the attack surface to an extent, that the applied mitigation can be bypassed [7, 15, 19, 20].

CFI offers a very different approach compared to randomization-based solutions. While CFI performs explicit checks, randomization-based solutions are *passive* and purely rely on a secret (e.g., the randomization offset in ASLR). The randomization-based solutions are typically more efficient. For instance, the average overhead of (fine-grained) CFI is about 21% [1], where randomization-based solutions typically only induce marginal or no overhead [39]. Isomeron induces a comparable overhead to CFI. This is because Isomeron is based on dynamic binary instrumentation, where CFI uses static instrumentation. As noted in Section V, dynamic instrumentation is preferred over static instrumentation to satisfy our requirements (e.g., disassembly coverage, dynamic insertion of instructions).

Concurrently to our work other mitigation techniques that aim to prevent JIT-ROP have been published. eXecute-no-Read (XnR) by Backes et al. [4] emulate execute-only code pages to limit JIT-ROP's ability to disclose code pages by keeping only *n* pages readable and executable at the same time. All other pages are marked as non-present and therefore generate a fault when read by the adversary. The security of XnR is based on the assumption that an adversary cannot predict the exact point in time when a code-page is readable. This makes XnR potentially vulnerable to disclosure attacks where Isomeron explicitly assumes that the adversary can disclose all code pages.

With Code-Pointer-Integrity, Kuznetsov et al. [27] aim to prevent control-flow hijacking by ensuring pointer integrity. This is achieved by using static analysis to identify all critical program variables that contain code pointers or pointers that eventually point to code pointers. At runtime, the memory for the variables is split into two areas: a secure area that contains all critical variables and a normal area for all other variables. All accesses to the secure area are instrumented to perform extensive security checks, e.g., checking that write operations to a buffer cannot corrupt critical variables. In contrast to Isomeron, CPI requires a complex static analysis and the source code of the targeted application, which makes it impractical for dynamically generated code.

Mohan et al. [34] combine corse-grained CFI with code randomization. Opaque CFI (O-CFI) uses static analysis to identify the destinations for each indirect branch and inserts CFI checks to limit each branch to the boundaries imposed by the lowest and hight address of all destination addresses. The bound range varies due to code randomization and the bound information are protected through segmentation which makes O-CFI immune to certain kinds of memory disclosure. O-CFI requires a precise static analysis to reconstruct indirect branch addresses which is an error prone task and is not compatible with dynamically generated code.

IX. Conclusion

Just-in-time return-oriented programming (JIT-ROP) has demonstrated the limitations of fine-grained code randomization schemes to protect against sophisticated runtime attacks. We show that defending against JIT-ROP is a challenging task, in that one can trivially extend the JIT-ROP framework to bypass a recently proposed JIT-ROP mitigation scheme. Not wanting to stop there, we then provide an alternative approach, called Isomeron, that not only randomizes the code but also the execution path. The idea is based on randomizing the execution path between two differently structured but semantically identical application copies. We developed a dynamic binary instrumentation framework for Windows and outlined the technical challenges. To further improve the performance of our technique, we are currently integrating known optimization techniques from existing dynamic binary instrumentation frameworks to enhance the efficiency of Isomeron and designing a compiler-based solution.

X. Acknowledgments

We would like to thank Stephen McCamant for his in-depth feedback that guided the paper's final revisions. We also thank Per Larsen and Ferdinand Brasser for fruitful discussions. This work has been co-funded by the DFG within the CRC 1119 CROSSING.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):365, 1996.

[3] M. Backes and S. Nürnberger. Oxymoron - making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

[4] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[5] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[6] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[7] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[9] X. Chen. Analyzing the first ROP-only, sandbox-escaping PDF exploit. http://blogs.mcafee.com/mcafee-labs/analyzing-the-first-rop-only-sandbox-escaping-pdf-exploit, 2013.

[10] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[11] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[12] M. Conover and w00w00 Security Team. w00w00 on heap overflows. http://www.cgsecurity.org/exploit/heaptut.txt, 1999.

[13] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[14] L. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can - secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[15] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[16] T. Durden. Bypassing PaX ASLR protection. *Phrack magazine*, 11(59), 2002.

[17] Gadgets DNA.com. How PDF exploit being used by JailbreakMe to jailbreak iPhone iOS 4.0.1. http://www.gadgetsdna.com/iphone-ios-4-0-1-jailbreak-execution-flow-using-pdf-exploit/5456/, 2010.

[18] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.

[19] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[20] E. Göktas, E. Athanasopoulos, C. Heraklion, G. M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter. In *USENIX Security Symposium*, 2014.

[21] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy*, 2012.

[22] M. Howard. Address space layout randomization in Windows Vista. http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx, 2006.

[23] Intel. Software guard extensions programming reference. https://software.intel.com/sites/default/files/329298-001.pdf. Accessed: 2014-11-16.

[24] N. Joly. Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php, 2013.

[25] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *IEEE Symposium on Security and Privacy*, 2003.

[26] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[27] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014.

[29] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[30] libdasm. libdasm. https://code.google.com/p/libdasm/, 2014.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[32] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security Symposium*, 2006.

[33] Metasploit. Metasploit. http://www.metasploit.com/. Accessed: 2014-07-14.

[34] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[35] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, 2009.

[36] R. Naraine. Memory randomization (ASLR) coming to Mac OS X Leopard. http://www.zdnet.com/blog/security/memory-randomization-aslr-coming-to-mac-os-x-leopard/595, 2007.

[37] N. Nethercote. *Dynamic binary analysis and instrumentation*. PhD thesis, PhD thesis, University of Cambridge, 2004.

[38] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[40] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[41] M. Payer and T. R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, 2010.

[42] A. Pelletier. Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit). http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php, 2012.

[43] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[44] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[45] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization.

In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[46] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[47] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[48] Solar Designer. Getting around non-executable stack (and fix), 1997.

[49] A. Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, 2007.

[50] A. N. Sovarel, D. Evans, and N. Paul. Wheres the feeb? the effectiveness of instruction set randomization. In *USENIX Security Symposium*, 2005.

[51] S. Sridhar, J. S. Shapiro, and P. P. Bungale. Hdtrans: A low-overhead dynamic translator. *SIGARCH Comput. Archit. News*, 35, 2007.

[52] Ubuntu Wiki. Address space layout randomization (ASLR). https://wiki.ubuntu.com/Security/Features#aslr, 2013.

[53] VUPEN Security. Advanced exploitation of internet explorer heap overflow (pwn2own 2012 exploit), 2012.

[54] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[55] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[56] Y. Weiss and E. Barrantes. Known/chosen key attacks against software instruction set randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[57] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium*, 2013.

[58] D. D. Zovi. Practical return-oriented programming. Invited Talk, RSA Conference, 2010.